Paradox Web Server

by Tony McGuire

Draft

last revised 11/7/2000
edited by Dennis Santoro

Introduction

I have been trying to figure out what level of "help" people are looking for in setting up Paradox as a web server. I finally decided that it really didn't matter. If I start with "up & running", those who weren't looking for this could ignore it and if I started with a technical explanation the same would apply.

So I am just going to start, and you can determine whether this is of any value to you.

Lesson 1 - The Basics

Paradox "talks" to the Internet in the same basic manner that any other program does. Until you actually make a request for, or are sending, data (and that is basically what all POST and GET actions are doing) there is no real action required by Paradox.

What I mean is that when you are on the Internet and click on a link, it is usually just going to result in a different page showing up on screen.

Where the exchange of data really comes in is when you get a block from a web page on screen where you type something in or select from a drop down or whatever, and then there is a button you click on; the information you filled in gets sent TO something. This something is Paradox (in these discussions, I am going to insist that Paradox do the work - not Cold Fusion, ASP, CGI, or anything else. I may occasionally make an exception but that will be clear if I do).

The block on the web page is called a field, just as in Paradox. The field is part of a form, just as in Paradox. However, the form is normally just a small portion of the page, not the whole page. On the Internet, a form is a formal structure, and can have drop down boxes, text fill-in fields, radio buttons, check boxes, etc.

A form (and most other elements, if proper HTML is used) has an open statement and a close statement. Within the opening and closing of the form, the elements describing the form (including the above field types) appear.

A simple form for collecting a name might appear as follows:

```
<form name="getname" method="POST" action="getname">
<input type="text" size="25" maxlength="23" name="fullname"> Enter Your Name
<input type="Submit" value="Send Name">
</form>
```

This would draw an input field, named "fullname", on screen 25 character wide, which would accept

up to 23 characters. Next to the field would appear the text "Enter Your Name". It would also create a button, with a label of "Send Name".

In the block, any 23 characters, up to the 23 limit, would be accepted. The extra 2 characters would remain as white space (I do this so that if the user continues to type beyond what I allow for the input they can see that not everything they typed was accepted).

When the user clicked on the button, the information that was typed into the block would be sent to the server that drew the form; along with the information filled in would be included the information that the form should be handled as a POST operation, by whatever the server understands to be the "getname" function.

Now, on the Paradox side, there is a method in a library that is designed to handle the input from this form (you create the method and the code that handles the input). The meat of the method might look something like this:

```
method getname(var Request OleAuto, var Response OleAuto) logical
var
namvar string
endvar

namvar=request.getfield("fullname")

;//request is an OLEAuto function that Paradox uses
;//for making the connection to the Internet (through the websrv.ocx).

;//Using request is how you are able to use Paradox to process
;//incoming "requests" for "action" from the Internet.

;//from this point, until you are ready to send something back
;//to the user's browser, everything you do is straight OPAL!!!

;//You now have whatever the user typed into the field on the form
;//on the Internet stored in the variable namvar.

;//If all you wanted to do was display on screen what the user typed, your next lines might be:

response.resultstring="You typed "+namvar
return true
endmethod
```

Response is the OLEAuto function you use to send text (Paradox's response) back to the websrv.ocx, which then sends it on to the user's browser. There is no need to create an HTML file, unless you wish. What you send is what the user's browser gets. Resultstring is where the response is stored and passed back to the websrv.ocx, which in turn sends it back to the user.

The response above is very simplistic. Normally, you will send the standard HTML constructs along with your actual response.

Most pages contain the following elements:

```
<html>
<head><title>This Page's Title</title><head>
<body>
</body>
</html>
```

Note the "almost" duplicated entries, such as <html> and </html>. These are the opening and closing statements of some element of HTML. Although you can get away, sometimes, with not closing an element, this depends in large part on how forgiving the browser that the viewer is using is. Then, there are a few elements that don't have the <element></element> construct. Not very many, though, and I will try to remember to point them out if I use any.

Next, let's assume that the form above is one that you use to gather the name of people visiting your site - you let anyone in, but only after they give you their name.

After namvar=getfield("fullname") in the method above you might include the following (assuming that you added tc tcursor to the var construct):

```
if namvar.size()<5 then
   response.resultstring="Name must contain at least 5 characters"
   return true
endif

;//5 is arbitrary. You could make this any number you want.
;//It just guarantees that the user typed something

tc.open("users.db")
tc.edit()
if not tc.qlocate(namvar) then
   tc.insertrecord()
   tc."name"=namvar
endif

tc."date"=today()
tc."time"=strval(time())
tc."logins"=tc."logins"+1
logvar=tc."logins"
tc.endedit()
tc.close()

;//Then, you might want something like this:

storagevar="Welcome "

if logvar>1 then
   storagevar=storagevar+"back "
endif

storagevar="to my world, "
storagevar=storagear+namvar
```

```
response.resultstring=storagevar
return true
endmethod
```

This would result in "Welcome to my world, John" or "Welcome back to my world, John" appearing on the user's browser.

What this does is maintains a list of your site visitors, the number of times they have visited and the last day & time they came back (I would set the "default value" on tc."logins" to be 0, so the field would never be blank even when a new record was created).

In this example, I opened the tcursor as I was processing the individual "request". This could just as easily have been a tcursor opened when the library opened; it could also already be in edit mode (I don't recommend the edit part, but experiment).

As you can see, this is all pretty basic OPAL. The difference is only the way you gathered information from the user - from an Internet form instead of from a desktop form.

The responses above would appear in very plain text, with no font control, alignment or coloring. Placing data elements, retrieving and displaying rows of records, font control, alignment, etc. will be covered later.

Hopefully, this lesson demonstrates just how simple the mechanics of retrieving data from the Internet is when using Paradox. Of course, we didn't get into the intervening mechanics - the step by step of the OCX and how it talks to Paradox. I did this on purpose. If you see how easy the meat of the process is, you'll be more likely to hang in there for the, to me, hard part. Not hard in making it work - Corel makes it pretty automatic and it works beautifully when you use the examples as a template. Most of us, however, want to know the inner workings so we can adjust them as necessary for the app we are setting up.

Lesson 2 - Retrieving Multiple Values

What follows is a quick & dirty example of retrieving a multiple of field values from an Internet form (taken from the Example that came with Paradox, with some of my own embellishments added).

Assume that you have an Internet form with 20 fields that you use to collect data from the people who visit your site. This can be any data type you want (even memo). The method below could be used to take the data and populate a table. The Internet form you are using to collect data from your site's visitors is not directly linked to the Paradox table, as a desktop form often is. The Internet form resides in the browser memory of your sites' visitors. Not until they click on the "submit" button is the link from their browser to your Paradox code established. Thus, you must first retrieve the information from the page the user was viewing and adding info to before Paradox can make use of the information the visitor typed in.

```
method getallfields(var Request OleAuto, var Response OleAuto) Logical
var
tc tcursor
dynvar dynarray[] anytype
```

```
i,n number
retvar string
endvar

n = Request.NFields - 1

;//Here, you are setting n to the highest numbered field on the form.
;//You subtract 1, because the first field on a form is 0. "for i from 0 to n" can then be used to
;//reference getfieldname() and getfieldbyindex() since this will then match with the starting
;// field number (0) that appears on the form, as well as total # of fields.

for i from 0 to n
dynvar[ Request.GetFieldName(i) ] =
ltrim(rtrim(upper(Request.GetFieldByIndex(i))))
endFor

;//request.getfieldname(i) is to get the name of each field and store it as the dynarray element name
;//request.getfieldbyindex(i) is to get the value of each field
;//This matches each field's name with its value.
;//You now have a dynamic array filled with field names and their values.
;//Now open the table that is to receive the data. In this case, when we created the form on the
;//Internet, we named the fields on the form with the same names as the fields in the table
;//which we will use to store the data that the user is inputting.
;//Further, the table we are using to store the data has a primary key on fields "username" and "address".
;//You would probably add a lot of validity checking at this point

tc.open("tblname.db")

;// this could as easily open on a secondary index as in
;//tc.open("tblname.db","scndindx")

;//and the qlocate() would apply to the secondary index
;//(if you needed to allow duplicates, for example)
;//at this point, your particular needs  (and standard Paradox/OPAL techniques) apply

tc.edit()
if not tc.qlocate(dynvar["username"],dynvar["address"]) then
   tc.insertrecord()
endif

if not tc.copyfromarray(dynvar) then
   ;//statement back to user
endif

tc.endedit()
tc.close()

;//build variable (retvar) containing return statement (in HTML) to user
;//this could be a "data accepted" statement, or even another form for them to fill in
```

response.resultstring=retvar
return true
endmethod

I did the qlocate() in the event the key field(s) were already in the table since then this would be a record update rather than a new record (standard technique, no different on the Internet than on the desktop). In the copyfromarray(), Paradox automatically matches the dynarray[] elements with the table's field names. If there are elements in the dynarray that don't match the table, they are ignored. If there are fields in the table with no matching dynarray element, the lack of match is also ignored.

Thus, with a few lines of code we have retrieved a number of data fields and posted them into a table.

How much easier can it get? Once you get past how to retrieve the data, everything is the same old (wonderful) Paradox you have been using. There are definitely some do's and don't do's that apply specifically to Paradox & the Internet (Paradox-I), and many tables will have to be restructured to make Paradox blazingly fast in the Internet world. But many of these concepts will apply to the desktop as well. I have restructured most of the tables I use on the desktop to take advantage of the techniques I used learning about Paradox-I.

Lesson 3 - The OCX and the Library

I have the beginnings of an FAQ on the technical side of request to response on the Internet.

Placing it on the Internet allows me to present this concept along with screen shots of the OCX and the Paradox form and library methods/procedures/const/uses sections.

I was a little leary of opening this up yet, as it is very much a work in progress and may be updated hour to hour. Also, it assumes that the viewer is a little more familiar with the OCX and the Paradox libraries than I would prefer to start out with. I think it may be helpful enough, however, that I am including a link to these pages: <http://www.instantmls.com/faq>

Please don't play in any other sections of the site or I will get skinned by my boss. The above is on a production server that we are relying on heavily.

Now, some more meat.

First, you need to go to the Examples ":wsample:" (C:\Program Files\Corel\WordPerfect Office 2000\Paradox\Websrv\Sample if you accepted the default) directory and open the Server.fsl Make sure you select the "edit the form design" radio button. Then open the Hercules.lsl file. If you did not install the web samples you can go back and install them by using the custom install from the CD.

When the <form> on a page on the Internet gets activated (we will work only with, and assume, POST requests), the request gets sent to the server indicated in the "action" portion of the form statement. If there isn't a full URL, your browser directs the request to the site that sent the form. Thus, you could actually "redirect" the action of the form to another site to handle the request if necessary, by putting the full "http" etc within the action statement. Otherwise, the location of the site that should handle the request is automatically included, along with whatever explicit action you declare. (e.g. if

the form was generated by <http://www.foo.com> and the action statement on the form was simply "getname", the form information would be sent to <http://www.foo.com/getname>. On the other hand, you could spell out in the form's "action" statement <http://www.redir.com/getname> and the form information would be sent to this other domain for processing.)

(Hint: I use POST rather than GET as the form method, since the GET command shows all variables being passed in the "Location" or "Address" bar on the user's browser. There isn't really anything confidential about this information, but it tends to be a little less confusing when the user doesn't see all the junk being passed. It also adds a little security, since a POST requires a form, whereas a GET can be typed into the "location" or "address" section of the browser. More on the security part later. This also simplifies programming, since the OCX/Paradox form then only watch for POSTs and process with OnPostRequest(). This is totally a personal preference, but it has worked well for me.)

OK. The POST has been sent from the user's browser to the location named in the "action" portion of the Internet form. Your OCX is set to watch for and process requests for POST actions by passing the information to Paradox. This kicks in the OnPostRequest() method (this method is within the OCX, not the Paradox form), which checks with OpenLibrary() (another method in the OCX, not Paradox form) to see if there is an associated library/function declared that should handle this request.

Here is where the flow of processing seems to get a little convoluted (seem convoluted already?) but in reality the flow is designed for maximum flexibility.

In the Example, there is a table named SERVER.DB. This table contains 4 fields. REQTYPE, which is "is this a POST or GET" type of request. URI, which is the action that is defined in the Internet form LIBRARY, which is the library file that should handle this request for action and HANDLER, which is the function (method) that should handle the request for action (remember, in lesson #2 I stated that the form "Action" and the method name didn't have to match? This is one place where you could change the method that should handle the action. You may want to open the SERVER.DB table and look at the records it contains to clarify this.)

Using a table to break the requests down and define a library/routine to handle the requests is not a requirement to making this all work. You could take several approaches. This is how the examples showed, and this methodology has worked very well for me.

OK. In the OCX, the VAR section defines a tcursor, and in the DoStartUp() method the tcursor is opened on the "server.db" table. This means that there is always a tcursor opened on the table that contains the information that the OCX will use to determine what to do with a request for action. The table is keyed on REQTYPE and URI, which makes it possible to use
qlocate(); this is a nearly instantaneous lookup.

When a request for action arrives at the OCX, and it is a POST request, the OCX activates the OnPostRequest() method. This method (OPR for reference) uses the open tcursor to check if there is a matching REQTYPE and URI. If not it does nothing but display a "404 - not found".

If it does find a match, it uses openlibrary() to ensure that the library defined in the matching record is either open or can be opened. Then, it passes to that library's "handlerequest()" method the "HANDLER" field in the matching record of the tcursor. This is why the uses block of the OCX does not have to contain every method within the library, or libraries, that it will be use. "handlerequest()" is the only method that will ever be referenced; this is also why "handlerequest()" MUST be in every library that will be used by the OCX. Through the above process, the uses block need reference only

1 method (handlerequest) and it applies to every library. This makes utilizing multiple libraries incredibly easy. And the "server.db" handles all the references; even though your site is "live" you can revise server.db and update your site on the fly. The same applies to libraries being used (the Paradox form with OCX has an "unload helper libraries" button for this specific purpose; with it the openlibrary() method is forced to reopen libraries, which causes code changes to take effect "live").

Next you have to have a library with the HandleRequest() method defined. In the sample Hurcules library's "Open" section, there is a declaration for a dynarray[] String. This dynarray[] accepts a split of the library's path on disk, and attempts to open a tcursor "tctemplate" (defined in the library's "var" section so it is available to all methods within the library) on a named .db file (in the example this is "hercules.db") that you use to store templates. If the tcursor can't be open, you get an error. In practice, you could eliminate this portion if you don't use templates of HTML; in reality you will probably rely heavily on templates. Templates, as utilized in the examples, are stored HTML that you can use over and over. For instance, I use these templates to store a common "look & feel" for the headers of various pages. You can also store portions of pages (such as a disclaimer that can be used specifically as needed), or drop-down lists that are common to several pages on your site. While the OCX also has a tab that allows "header" and "footer" pages that it automatically adds to every page it pushes to the user, I don't necessarily want exactly the top and bottom of every page to be the same. This is where templates allow more flexibility.

Anyway, with the tcursor defined at form level, and opened when the library opens, a defined method (fetchtemplate() in the example) can be used with the various methods in the library to retrieve a specific template. The hurcules.db (stored templates) has 2 fields
1. Template. A primary key field with the name of the template
2. HTML. A memo field that contains the HTML that makes up the template.

In the "const" section of the library, there are declarations for "1" and "2". These represent "field #1" and "field #2" in the hurcules.db table. Fetchtemplate() then uses these constants to look for the named template in field 1, and sends back the content (memo field) of field #2 if it finds a record with matching "template" field. Thus, a declaration of the fetchtemplate() method might be retvar=tctemplate.fetchtemplate("tony"). This would do a qlocate() and return the HTML field contents, storing them in retvar (I use retvar as a standard variable declaration in most methods - "ret"urn "var"iable from the old Paradox DOS days).

Now, the real meat of how a request is passed to, and the response passed back from, Paradox:

The HandleRequest() method is the start of EVERY call to a library. Its declaration is:
method handleRequest(strHandler String, var Request OleAuto, var Response OleAuto) Logical

strHandler contains the information from Server.db's HANDLER field, passed from the OnPostRequest() method within the OCX I have no idea why the "Request OleAuto" and "Response OleAuto" are part of this declaration; I have always guessed that they create handles back to the OCX, and therefore to the Internet. I do know that it works. (Perhaps someone out there could shed some light on this.)

Within the HandleRequest(), I use (and so does the example) a switch.....endswitch construct to determine which method within the library should actually handle the request.

Using the original "getname" from the Internet forms "action" statement, and assuming that the "handler" field in "server.db" matched with this, HandleRequest() might contain the following:

```
switch
case strHandler="getname"          : return getname(request,response)
otherwise                          : return notfound(request,response) ;
endswitch
```

notfound() is a custom method with a message I wish to display when I don't have any other method defined to handle a request

What this apparently does, using request, is create the link from all data available from the Internet form to the Paradox method. All other data (browser version, IP address, etc.) normally available about the browser/machine is also available to the Paradox method defined here (getname).

This is where lesson #1 comes in. Remember

```
method getname(var Request OleAuto, var Response OleAuto) logical
var
namvar string
endvar
namvar=request.getfield("fullname")
```

Yep, HandleRequest() has activate this method within the library. getname now has access to the data being sent by the browser making this request. The form data, as well as a slew of information about the browser and machine making the request.

As in example 1, everything done within getname() is plain OPAL, except the request.getfield() and response.resultstring="" statements. (There are others, but I don't want to cloud the issue any more than necessary. You can find more info on this in the pdxinet.hlp file.)

Once you process the incoming data from the form (in this case, the data from the "fullname" field), and take any other actions you wish, the information you wish to go back to the user is sent using the "response" OleAuto function as far as I can tell.

```
response.resultstring="whatever you want to send back to the Internet user
goes in here"
return true
endmethod
```

I hope this explanation really did flow. It may be an oversimplification to some of you, and it may have confused some. I realize that I got off-track with the fetchtemplate() stuff, but this is pretty important for allowing flexibility and reuse of HTML code. Also, while the other methods in the example library can be dumped, handlerequest() and fetchtemplate() should not, in my opinion. Handlerequest() is absolutely required (although you could rename this to anything, as long as all references in the OCX were changed), and fetchtemplate() is just too darned useful.

Please try to keep in mind that I didn't create any of the processes described in these "lessons". Paradox came standard with every bit of it. Whoever figured this all out gave us a pretty darned flexible system for creating database-driven web sites. They just forgot the roadmap.

Lesson 4 - Accessing Paradox Data

Let's recap with a breakout of the various elements of a Web site run by Paradox, using the Examples that came with Paradox as a starting point:

1. You need a Paradox form with the OCX "installed".

2. You need a Paradox table that matches the incoming request's "action" statement (function) with a library, and the method that will be executed by Paradox to fill the request.

This table's 4 fields are

REQTYPE (P or G, for POST or GET)
URI (Action element from the Internet from, preceded by a slash "/")
LIBRARY (The library Paradox should go to and pass the HANDLER to that library's handlerequest() method)
HANDLER (The actual method you are using to process this URI. This can match the URI (without slash), or be anything else you wish; it just has to match up to one of the case statements with the library)

3. You need a Paradox library that contains a handlerequest() method. In the handlerequest(), you will have a switch...endswitch construct that matches the name passed by the OCX to a method in the library. The name passed by the OCX will be the entry in the table's "HANDLER" field (#2, above). You can match the case statement to the internal library method or change it; personal choice. Only real reason I found not to match these was if you wanted to add a little confusion to anyone trying to figure out your coding practice and program flow.

4. If you want to use "templates", as in the example, you will need a table of templates. These templates are HTML code you will use multiple times throughout your web site(s). If you use the fetchtemplate() method included in the examples, this template file will contain 2 fields:

TEMPLATE - A primary key field whose value you will pass to fetchtemplate() (as in fetchtemplate("GETHEADER")). fetchtemplate() will return the code in the second field

HTML - This field is a memo field that contains actual HTML that you want to make "reusable".

Because you are using Paradox as your Web Server, references to tables within your library can use aliases. This is very handy, since a web visitor can only "see" files in the "root" directory and below. Using aliases, you can place the data you are searching (including the above tables/libraries), anywhere, as long as Paradox has access to it. Thus, you could actually search against data on a completely different machine than the one Paradox and the Websrv.ocx are executed on (An alias would take care of it, just as on a LAN).

I have found it very helpful to declare database variables at the library level, then "open" these databases in the libraries "open" method. This speeds things up considerably when coding; tcursor.open("tblname.db",databasevariable). If you need to change which alias a database variable is opening, you only need to change it in the library's "open" method (of course, you can also change where the alias is pointing, but I am assuming the original alias needs to be maintained - I ran into this scenario, and it caused me no end of tedious recoding).

Opening the tcursor, you can include the database name as well as a secondary index. Since just

about everything I do on the Internet makes use of secondary indexes, I incorporate this into the coding of almost every method.

The next statement after opening the tcursor is the tcursor.setrange(). In two steps, I have most of the answer I need, if not all of it.

If the setrange() incorporates all of my search parameters (let's say someone searching for a particular model of car (Corvette) offered within a certain price range ($15,000 - $18,000), the setrange() results in a tcursor containing the complete set of records that need to be returned to the user.

Assume a method for searching for a car. Define the variables to be used:

```
var
tc tcursor
cr string ;//car model
lp string ;//low price
hp string ;//high price
ly string ;//earliest year built
hy string ;//latest year built
da dynarray[] string ;//dynarray for additional search criteria
retvar string ;//string variable to store line of return HTML code
endvar

cr=request.getfield("model")
lp=request.getfield("lowprice")
hp=request.getfield("highprice")
ly=request.getfield("lowyear")
hy=request.getfield("highyear")
```

Thus

```
tc.open("autosforsale.db",databasevar,"carprice")
tc.setrange(cr,lp,hp)
```

is the entire code that narrows down a table to those records that match the "search" criteria of car model and price range. It does so in less than 1 second (much less!) at least on a table containing 40,000 records in this case.

Of course, this results in all Corvettes in this price range being included in the tcursor.

What if the Internet visitor only wanted those Corvettes made between 1968 and 1973?

That's where setgenfilter() comes in. Setgenfilter() respects the limited view of a setrange(), using a dynamic array to pass the additional criteria to setgenfilter().

What you can do is define a dynarray[] in the method's var...endvar statement (just as you declared the tcursor, model, low price and high price variables)

Set up the criteria for the setgenfilter() using the dynarray:

```
switch

case (ly.size()>=4 and hy.size()>=4) : da["yb"]=">="+ly+", <="+hy

case ly.size()>=4 : da["yb"]=">="+ly

case hy.size()>=4 : da["yb"]="<="+hy

endswitch
```

("yb" is the field name in the table that contains the year that the car was built. This causes the da["yb"] criteria to automatically be applied against that field in the table. Yes, I realize that this is standard OPAL, but that's what I have been saying all along. I just don't want someone to get lost in all of this because I didn't include an explanation. I didn't understand setrange()/setgenfilter() until I started programming for the Internet in Sept. '99, and I don't want to take for granted that I am the only person who didn't understand this concept. Then I'd really feel dumb)

Now, you have a criteria for the setgenfilter() built using the dynamic array. To apply it

```
if da.size()>=1 then
tc.setgenfilter(da)
endif
```

You can use multiple dynarray criteria elements within 1 setgenfilter command; just add additional da[""] elements before executing the setgenfilter(da).

I use the if..then..endif construct in the event year didn't matter to the person doing this search. While model and price range are required, year built isn't for this example.

Setrange() narrowed things down to only Corvettes in a certain price range nearly instantaneously becuase it uses indices. Now the setgenfilter() only needs to be applied against this much smaller data set. This results in the setgenfilter() being nearly instantaneous as well in this example, anyway. Your data/secondary indexes will dictate how small of a data set setrange() will result in, which will dictate how many records setgenfilter() will be applied against.

As you might imagine, table structure is absolutely critical to this model. There are no joins or calculations (calculations are possible, more on this later). And everything hinges on the secondary indexes. I restructured just about every table I use for the Internet, although there are ways around this in some cases. I started out with a separate file for "searches", with the complete records maintained in their original tables. My situation is a little unique, however, in that every record has a unique reference ID; I can do a qlocate() of the reference ID on the tables containing the full data. Not all of your situations make this possible.

Lesson 5 - HTML to Paradox and Back

This next part is where I struggled hardest. For some of you, this may be where you say "No Way". Once you "get" it, learn how helpful copy & paste is, and learn to use those Templates I keep mentioning it isn't bad at all. You just have to program in 2 languages at the same time: OPAL and HTML.

This is also why I have become friends with HotMetal (www.softquad.com). It taught me to create HTML that is available to virtually any browser out there. Too many sites want to use the latest and greatest code possible; this is fine if you don't mind locking out the millions of AOL users and many Netscape users, as well as a large portion of the MAC community. While Microsoft says "So what, buy mine", I'm not ready to limit myself that much.

OK. We worked out in Lesson 4 how to narrow a table to a set of records that need to be sent back to the user. Please understand that you will not be sending the Paradox table to the user. You will be sending the data from the Paradox table in HTML format, and using HTML formatting codes/constructs to present the data to the end user (Internet Visitor).

Let's start with the structure of your data, containing information about cars.

Field Data

Model alpha
Price numeric (not money!)
YB numeric
Color alpha
Mileage numeric
Owner alpha
Phone # alpha (to call owner)
email alpha

I probably left out some critical stuff, but I am not in the auto business. Let's just say that this is the information you gather from your clients selling cars, and then make available to people who want to buy them. Much of this stuff is specific to cars, but you could make this table generic (so it could handle many different types of items for sale) or you could have a different table for each type of item (the first would be easier to manage, the second would make your site faster due to smaller searches). I take the second route every time. Through coding the table management becomes less of an issue. You can make it manage itself. For example, in a form on your site that retrieves new entries the person entering the data selects the type of entry they are making. This drop down becomes the "action" statement in your Internet form. This automatically causes Paradox to process this entry based on the type of entry it is based on the information in the server.db entries and underlying library.

Having applied the setrange() and setgenfilter() from Lesson 4, let's say that we have a table with 15 records (yours is a nationwide site, after all).You now need to send the record set to the user. The way I handle this is normally by starting with a fetchtemplate()

retvar=fetchtemplate("carresultsheader")

In the template table, the HTML field might look like this:
<html><head><title>Cars that match your search criteria</title></head>
<body>

What this does is retrieve the above HTML into the "retvar" variable. <html> tells the user's browser that the data they are receiving is HTML. The text between the <title></title> will appear in the user's browser as the title of the page they are viewing (top right of the browser, on the "blue" app title bar).

Since this is a template designed specifically for setting up the return results from a search for a car, you could incorporate even more HTML into the template since every result would start the same way:

```
<table width="100%">

<tr>
<td>Model</td>
<td>Price</td>
<td>Year Built</td>
<td>Mileage</td>
<td>Color</td>
<td>Owner</td>
<td>Phone</td>
<td>E-Mail</td>
</tr>
; note that we do not close the <table> construct
```

I placed each of these opening/closing constructs on their own line only to make it a little easier to read. The <table> statement declares an HTML table for holding rows & columns of information (think of it as a spreadsheet). <tr> declares a row within the table. <td> declares the start of a column (cell). </td> closes the cell. </tr> ends the row.

What this accomplishes is a "header" row for the data below it. Now, we need to get the data from the table and put it into matching rows (a record) and columns/cells (each field of data).

```
tc.open("autosforsale.db",databasevar,"carprice")
tc.setrange(cr,lp,hp)
scan tc :
retvar=retvar+"<tr>"
retvar=retvar+"<td>"+tc.model+"</td>"
retvar=retvar+"<td>"+format("e$c",tc.price)+"</td>"
retvar=retvar+"<td>"+strval(longint(tc.yb)+"</td>"
retvar=retvar+"<td>"+strval(tc.mileage)+"</td>"
retvar=retvar+"<td>"+tc.color+"</td>"
retvar=retvar+"<td>"+format("cc",tc.owner)+"</td>"
retvar=retvar+"<td>"+tc.phone+"</td>"
retvar=retvar+"<td><a href=\"mailto:"+tc.email+"\">"+tc.email+"</td>"
retvar=retvar+"</tr>"
endscan
tc.close()
```

The scan, of course, steps through each record of the limited view table, and applies the code between scan....endscan

retvar=retvar+".." results in the individual lines being added to the string variable retvar. This is very handy. You might be tempted to make one long line out of this. But remember that a string variable is very limited in size as to each declaration; by breaking the lines down, you are only limited by the total character count of the string variable.

The end result of this is that each field of the table is placed in a column that matches the header you

got from the fetchtemplate() command. By starting the return from each record with "<tr>" you are creating a new row, just as in the Paradox table. Using "</tr>" declares the end of the row, thus matching each row in the Internet table with a record in your Paradox table.

The "mailto:" in the scan provides some additional functionality. What this does is not only display the content of the e-mail field, but makes it "clickable". The results is that the viewer can click on the e-mail on-screen and their e-mail client automatically pops up with the e-mail addressed to this owner (from the table's e-mail field).

Also, the width="100%" makes the HTML table automatically size to the full width of the viewer's screen. It widens/shrinks the columns of data to fit the screen of the person viewing the data.

There are individual commands that can be added to the <td> declarations that cause the cell to be aligned left/center/right, change font size/color/bold/etc. but you'll have to discover these.

When you close the tcursor, the next thing you will likely do is declare the end of the Internet table. You do this with:
retvar=retvar+"</table>"

Now, you have the information your user asked for, formatted for a browser, that the user can make some sense of. It is nicely formatted in rows and columns. Next, you need to declare that you have reached the end. You can always add as much as you want by adding retvar=retvar+".....", but we'll call this the end of our return page.

You could use another fetchtemplate() command to get the last bit, but there is so little that I usually include the ending code in the method I am using to get the data and return the results:
retvar=retvar+"</body>" ; close the body of the HTML page
retvar=retvar+"</html>" ; let the browser know that this is then end
response.resultstring=retvar
return true
endmethod

That is the whole thing! We just got the user's search criteria (Lesson 4), did a "search" against a Paradox table, and returned the results back to the person on the Internet who made the request. NOTE: I did not open a textstream and create an HTML file. The return data gets sent directly back to the user's browser. There are no files to maintain. The hard drive gets accessed in order to retrieve the data, but not to create a file, nor for the user's browser to read from. This isn't desirable in every case; you may want a "permanent" for future reference. If this is the case, then you can always open a textstream (naming it based on whatever schema you want), then use something like:

textstreamvar.writeline(retvar)
textstreamvar.commit()
textstreamvar.close()
response.resultfile=file_name_you_opened_the_textstream_with
return true
endmethod

If you want the underlying data you send to the user to appear nice (in case they look at the source

code), add \n wherever you want a line break (new line) in your HTML code (I usually do this at the end of each record; thus </tr> becomes </tr>\n).

Q & A

I got the following as suggestions/questions. I will try to answer as best I can:

Q. describe what the environment is that your system works in.
A. I have a single copy of Paradox with OCXs (currently 3, was 4) running on a Windows NT 4.0 system. We host the server ourselves with a T1 connection (I work for a very progressive company). The server is a 550Mhz system, 256 MB RAM, and RAID 5 SCSI disks. (to support the outside companies we will be hosting, we just purchased 2 700Mhz dual CPU systems specifically to place their data/sites on; the new machines may support more users in the long run, but Paradox doesn't answer requests noticeably faster [how fast can you get?]). I have made this work on a Windows '98 machine with 128MB RAM.

Q. What I mean is, where is the PDOX table?
A. The Paradox tables reside on the same machine, in a directory "above" where the form with the OCX is executed from. As long as Paradox can "see" the table, this location can be anywhere. Obviously, the system is faster if Paradox is on the same disk as the Paradox tables. More, you could use an SQL system, and have Paradox hitting a separate box with Oracle, MySQL or any other that Paradox can talk to running on the separate box. On the wild side, you could have a wide area network where Paradox was on a server in Sacramento searching against a Paradox table on a server in Boston.

Q. Are we talking about using Apache or PWS?
A. No, no Apache or Personal Web Server required. Since the OCX is a Web Server, nothing else is needed. Paradox in essence becomes a Web Server by virtue of running the OCX in a Paradox form. This is one reason why you could run a site from a machine with only Windows '95 or Windows '98 installed. To speed things up, and enhance overall performance, I now have the OCX only taking POST requests and passing them to Paradox. Requests for HTML files on disk, as well as for images, are now being handled by IIS. This is only because IIS is free with NT server. All of these requests were initially being handled by the OCX. I broke the actions apart as a sort of "poor man's load balancing". And it works beautifully.

Q. Can this be done with an ISP?
A. Yes. It could be. Unfortunately Paradox has a fairly large overhead, so few ISPs would let you run a full copy of Paradox on one of their boxes. And, since they would have little-to-no say in what you did, you could potentially bring their whole system down. Not a situation many ISPs would put themselves in. However, if you have an ISDN, frame, cablemodem, T1 or above, and have a static IP (which is really all you need; this could work if you use a modem but get a static IP when logging in) you can host your own site without asking anyone anything. Have a domain name pointed to the box with the static IP, load a copy of Paradox, create your site and you are hosting your own site. Sound simple? In the overall scheme of things it is. Once I finally understood what the examples were trying to tell me, I did what I have just described. I pointed my browser to my machine's IP address and there was the site in the examples. I went to another machine and pointed its browser to my machine's IP and it showed the example site as well. From there, I started playing with the tables and libraries in the examples and watched what happened when I pointed my browser there. Of course the whole

situation is now far more complicated. But not as different as stepping up from what I am now working on to a site the size and complexity of a Yahoo or e-Bay. If I needed to support sites of this size & complexity, I would probably move to a different tool than Paradox. But if Paradox will do the job I need, why spend the time & money learning something that isn't really needed?

Q. How does an ISP play into this?
A. Doesn't, in most cases. Few ISPs will want to install Paradox on one of their servers. A "colocate" might work; there you supply the machine & software, but take advantage of the ISP's connectivity.

Q. Is Paradox and BDE installed on my server or ISP server?
A. If you have a server, you can become your own ISP. No need to pay someone else at that point (other than for connectivity and someone to keep it all running).

Q. Same server as my web pages?
A. Normally, this would be the case. Your (or ISP's) server would have the whole tamale on the one machine. As your needs change, you can add a server, bring the whole thing in-house and increase connectivity, or several other scenarios. You would likely update the data residing on an ISP's server using FTP. If you host your own, the data could be updated from an desktop machine with a copy of Paradox installed (at this point, the situation would be like a client/server situation; one of the clients accessing the information would be the web server.)

Q. Can you explain how all this might work if a company is using a third party ISP?
A. If you can get the ISP to put Paradox on one of their servers, it could be run off of their server. This isn't likely. You could also purchase a server, and locate it at the ISP (colocate). In this way you don't have to pay for a higher speed connection (although T-1 prices have decreased dramatically), you use the ISP's connection. The second scenario is far more likely an option; aren't very many ISPs who would work with you on the first option. Colocate costs are pretty high; the numbers I have been quoted are anywhere from $250/month to $450/month. We get our T-1 for ~$650/month. The bandwidth is shared by the websites and our entire administrative staff (about 20 people right now).

Q. Maybe some more general facts about how different connections to the web matter.
A. The real difference you run into with types of connections (T1+, frame, ISDN, cable modem, DSL etc.) is speed; in some areas reliability may also be an issue. We started out running the whole thing on a server connected to the Internet on a 256k ISDN connection. And the connection was also shared by 11 people in the office. It still ran blazingly fast, with some slowdowns during the peak periods. Cable modem is a higher speed down ("to you") than up ("from you"), as is DSL. Which means that, since the web server is receiving on the down side and sending to the up side, it would be receiving requests on the higher speed and sending responses on the slower speed - not a good scenario in my opinion. Frame, I believe, is available in speeds up to 384k (maybe 512?), which would suit many sites. ISDN, I think, is only up to 256k. If you go this route, make sure your ISDN connection is "always on", or your site will experience long delays at times (some are a "wake up on initial request" connection). For a smallish site, ISDN would probably work fairly decently. Frame would probably be a better option, though - I think a router can be purchased that would support frame now, and also handle a T-1 if you found you needed it later.

Host your own. Get started with ISDN or Frame, and move up to T-1, T-3 or Fiber Optic as necessary. All it takes is an "always on" connection, and to have the domain name you will be using pointed to the (static or permanent) IP of the computer where your site is located. Your ISP can do this on their equipment (DNS server, which also tells the rest of the Internet where your site is located). They

shouldn't even charge you anything extra.

Take the hardware stuff above with a grain of salt. I am a programmer, and leave the "hard" stuff to those who can handle it. What I have learned has been kind of by osmosis, listening to others discuss how they would handle what I was asking for.

One item I didn't address.
The copy of Paradox that runs a web site is a standard copy of Paradox, installed just as you would for your desktop, on the machine that will host the web site.

I do boost the BDE memory items and file handles, but that's it. Otherwise, it is just a standard Paradox install.

I have even gotten this to run on Paradox Runtime, although registering the OCX with Runtime was a bugaboo. Don't remember how I got this to work, but I did. So you could provide this to your clients to run their own, as well. (As far as I am aware, the OCX is "redistributable".)

In addition to the above, there was a comment:

"The downside is that one has to develop a whole new application"

This is in many ways true. The interface is definitely different. You have to re-create the look & feel of your app using the language of the Internet - HTML. That's why I use HotMetal when getting started on a site or function within a site. I create everthing in a GUI environment, then copy the resulting code into Paradox. Hand coding HTML isn't the most fun I've ever had, so I cheat whenever possible.

The underlying application code, which in many cases was taken care of by Paradox automatically, must be written to control the flow of data in both directions (request & response), at a much more basic level than is required on the desktop. Field validation must be taken care of by offering selections you define, or through some process that guarantees the data is valid for the data type you are working with (Lord knows you can't count on absolute accuracy in data entry from the world at large; heck, I have even made a couple of mistakes). But the language you use to do this is already a part of your vocabulary (OPAL). And HTML is really a pretty easy language to grasp. Or at least the most common elements you will likely be using.

"The upside is that it works"

Yep. And it works incredibly well, with blistering speed. Our company has been radically changed because we now have access to information anytime and anywhere we can get to a browser. Not just at work - at home, at our clients' homes, and we can even search for properties from a CELL PHONE. One agent sold over $300,000 worth of property in a week because he had the ability to search for property from his clients' cars as they were driving around. And PARADOX is the reason this was able to happen. The cell phones talk directly to Paradox (via the OCX), and Paradox (via the OCX) sends the results directly back to the cell phone. How many sites have you heard of that actually make THAT happen. Ours has been running for a couple of months. Without Paradox, I doubt I would have been able to figure this out and make it work. We are now using Paradox (the same copy) to support outside companies. They have access to their local MLS properties, either by browser or CELL PHONE.

These are valid "objections" (although I think they were brought out to make the whole picture clear to everyone, rather than as objections). If we don't know the whole story, it makes it much harder to make good decisions.

I AM VERY GRATEFUL that the above were brought up. No doubt if someone was willing to ask these questions, there are more than a handful who want to know the answers.

No one will ever convince me that Paradox is not THE BEST way to get started when the information to be placed on the Internet comes from a desktop Paradox system. It WILL NOT support you if you are on the level of an E-Bay or Yahoo. NOTHING will support you better if you are a small to medium company with information you want to make available to the world (or just your own employees).

More

One thing I forgot to add on to the add-in.

This entire process, running on a fairly fast server, takes less than 1 second.

It can definitely take longer, depending on the complexity of the "search". (And seem longer, based on the User's Internet connection.)

Using our sites as an example, retrieving 15-20 records, or more, (from an 8,000+ record table) with an area/price search (a single setrange() after the library code first checks the requests for field validity) is a "press enter" and "see the results on screen" type response. This is when I am doing the search from my desktop (both ends are a T1). From a 28K modem, this take 3-4 seconds. In both cases, Paradox retrieved the data set in less than 1 second. The difference is how long it takes based on the user's connection to the Internet. In either case, the time difference is handled by the OCX. Paradox has moved on to taking and filling other requests.

If Paradox takes a while to process and return a page, the OCX "holds" any requests that come in until Paradox is free to handle them. The OCX is multi-threaded, where Paradox is single-threaded.

Feedback

Again, please speak up if you have any questions, comments, or a different way to apply this stuff. There may well be a lot of better ways to put the stuff Corel gave us to better use. _ALL_ ideas welcome. I am more than willing to learn to make this stuff work better.